Lab 1, Part 1: Java and JSON and. . . ?

Due date: January 10, 11:59pm.

Lab Assignment

Assignment Preparation

This is a pair programming lab. You are responsible for finding your teammates for this assignment, and you need to do it fast. I will assign partners for everyone who has not been able to find one.

For this lab, and for Lab 1-2 you will remain in the same pair.

The Task

In this class, a healthy dose of programming will be done in Java. We will also be working with a variety of input data, but a lot of the time, the data will be stored in JSON, a popular lightweight data format.

In this lab, you will practice the use of Java for handling JSON objects and collections of JSON objects.

For this lab, you will perform two subtasks:

1. **Data Generation.** You will write a series of programs that generate random data (in some pre-specified formats) and dump this data into files.
2. **Data Consumption.** You will write a series of programs that consume the data produced by your data generators and compute certain infomration about the consumed data collections.

   Lab 1-1 is the data generation task.

   Lab 1-2 (to be assigned during Thursday, January 7 class) is the data consumption task. Lab 1-2 will be due beginning of the lab period on Tuesday, January 12.

Java Support for JSON

JSON[1] (JavaScript Object Notation) is a lightweight text-based format for representing structured and semi-structured objects and their collections. It is a somewhat less bulky (albeit not as powerful) format than XML, but JSON documents and XML documents share a lot in common and can be translated into each other.

Java has ample support for JSON. http://www.json.org lists 26 different Java libraries for parsing and generating JSON objects. In this lab you can select any of the JSON libraries for Java.

In preparation for this lab I used one one of the libraries, `org.json`. I have made the `.jar` file for the library available from the course web page. I have also made a couple of small programs for parsing JSON and generating JSON objects available.

Data Generation: Data Formats

In this assignment, you will "spoof" JSON objects generated by two applications: an on-line game `BeFuddled`, and a text messaging service `Thghtshre`[2].

Data Format for BeFuddled on-line game

The BeFuddled on-line (and mobile) game is a simple puzzle game that is played on a 20×20 playing field. The exact rules of the game are largely irrelevant[3], but what is relevant is how the web-based application and the mobile application record the data about the game.

BeFuddled is a turn-based game. On each turn, the game player can take one of a few actions (outlined below). Each action, when completed, generates a log record that uniquely identifies the user, the action, and the context for the action. In a real scenario, the log record would be much longer, but for the purposes of this particular task, we will use somewhat streamlined log records.

To play BeFuddled one must create a simple user account. No anonymous play is allowed.

A single log record has the following JSON format:

```
{
  "user": <userId>
  "game": <gameId>
  "action": <actionObject>
}
```

[1] http://www.json.org/

[2] Both are, **I hope**, fictional

[3] I.e., I had not had enough time to invent a completely new game for your amusement.

Here:

`<userId>`:	Unique id of the user playing the game.
`<gameId>`:	Unique id of the game being played.
`<actionObject>`:	a JSON object (see spec below) specifying the action.

User Ids. All user ids have the same format: a string that starts with a lower-case letter `"u"`, followed by a number. Sample user ids are `"u1"`, `"u234"`, `"u120349534"`. There are no leading zeroes in the number part of the userid.

For this assignment, there are 10,000 user accounts for the game and their user ids are `"u1", "u2",..., "u10000"`.

There are four types of actions in the game that are recorded. They are listed below.

Game start action. This is the first action of every game. A game start JSON description **always** has the following format:

```
{
  "actionNumber": 1,
  "actionType": "GameStart"
}
```

Game end action. This is the last action of every game. The game end action records the final status of the game. The final status comes in two attributes: the number of points collected by the player, and whether the player won the game or lost it. A game end JSON object has the following format:

```
{
  "actionNumber": <actionNumber>,
  "actionType": "GameEnd",
  "points": <points>,
  "gameStatus": <WinOrLoss>
}
```

Here:

- `<actionNumber>` is an index of the action in this particular game. It is an integer number.
- `<points>` is the number of points the player scored in the game. It is an integer number.
- `<WinOrLoss>` is either `"Win"` or `"Loss"`.

Regular move action. A regular move action is a user clicking on a specific square on the 20×20 playing field. This action is recorded with a pair of coordinates of the selected square, and the number of points it brings to the user. The JSON object for a regular move action has the following format:

```
{
  "actionNumber": <actionNumber>,
  "actionType": "Move",
  "location": {
               "x": <XCoord>,
               "y": <YCoord>
              }
   "pointsAdded": <pointsAdded>,
   "points": <points>
}
```

Here:

- `<actionNumber>` is the index of the action in the current game. It is an integer number.
- `<XCoord>, <YCoord>` are the values of the X and the Y coordinates of the selected square. These are integers in the range between 1 and 20.
- `<pointsAdded>` is the number of points brought in by the move. This is an integer number that can range from -20 to 20.
- `<points>` is the total number of points the player has after the current move is completed. The is an integer number.

Special move action. There are four special moves in the game. Each move can be used exactly once per game. The move names are `"Shuffle"`, `"Clear"`, `"Invert"`, and `"Rotate"`. The moves bring in a certain number of points. THe JSON object documenting a special move has the following format:

```
{
  "actionNumber": <actionNumber>,
  "actionType": "SpecialMove",
  "move": <specialMoveName>,
  "pointsAdded": <pointsAdded>,
  "points": <points>
}
```

Here, `<actionNumber>`, `<pointsAdded>` and `<points>` are the same as in a regular move JSON object, while `<specialMoveName>` is one of `"Shuffle"`, `"Clear"`, `"Invert"` and `"Rotate"`.

Data Format for Thghtshre Messaging Application

Thghtshre messaging application is a simple mobile messaging application that allows its users to send messages to other users subscribed to their messaging feeds. For this particular application, we concentrate on generation of messages themselves.

A single message is encoded as a JSON object in the following way:

```
{ "messageId": <messageId>,
  "user": <userId>,
  "status": <messageStatus>,
  "recepient": <recepient>,
  "text": <messageText>
}
```

Optionally, the message may be made in response to another message, in which case, one more attribute:

```
 "in-response": <originalMessageId>
```

is added to the JSON object (typically, after the `"recepient"` attribute, but since JSON is not order-sensitive, it does not matter where it is placed).

Here,

- `<messageId>` is a unique Id of the message. It must be an integer.
- `<userId>` is the unique id of the user sending the message. User ids have the same format in `Thghtshre` app as they do in the `Befuddled` app. We also assume that `Thghtshre` has 10,000 users with the same user ids.
- `<messageStatus>` is either `"public"`, `"protected"` or `"private"`.
- `<recepient>` is either `"all"`, `"self"`, `"subscribers"` or any single user id.
- `<messageText>` is a short message consisting of a number of words. (See the rules for message generaion in the description of the generator).
- `<originalMessageId>` is an id of a message. It is an integer number that must be smaller than the the `<messageId>` number.

Here is a simple JSON object representing a message:

```
{
 "messageId": 204532,
 "user": "u4053",
```

```
 "status": "public",
 "recepient": "subscribers",
 "in-response": 204302,
 "text": "I do not think so!"
}
```

Data Generation

For each of the two applications you will write a simple data generation program. The primary inputs to the data generation program will be the name of the output JSON file, and the number of JSON objects to generate. The programs will take the input and will generate the appropriate number of correctly formatted (according to the data description provided above and the generation rules provided below) JSON objects. The objects will be saved into the file with the given file name.

Each of the two generation programs will be required to abide by a special set of object generation rules designed to make your generation procedure provide reasonable approximations of real activity.

The generation rules are outlined below.

Please note, that this part of the assignment is essentially an exercise in the creative use of the random number generator. There is a number of different ways in which each JSON object generator can be constructed that will yield correct data. Most of the specifications below are declarative, that is, they explain, what a correctly generated sequence of objects should look like, and what sorts of objects violate correctness conditions. I do not provide algorithmic instructions here, but I am happy to discuss them with any interested team. The actual generation procedure is not really a secret I am hiding from everyone.

Data Generation for the BeFuddled Game

The general idea behind generating the JSON objects for the `BeFuddled` game is that we want to make sure that the generated objects, when viewed as a log of activity for `BeFuddled` players actually make sense.

The rules are:

R1. Each player can only play one `BeFuddled` game at a time. Players may play different games throughout the given time frame, but they must complete their previous game before starting a new one.

R2. Game Ids must increase over time. While strictly speaking it is not necessary to record every game, if you record an action of starting game number 45, you cannot start game number 32 after it.

R3. Only one player can play a game with a given number. That is, all records for a game number X must have the same user id.

R4. First record of every game must be the "GameStart" action JSON object.

R5. Last record of every game must be the "GameEnd" action JSON object.

R6. Action numbers within a single game must form a non-decreasing sequence, and all action numbers for a single game are unique. (For simplicity you can just generate action numbers 1,2,3,... for each game id, although, strictly speaking it is not required, we simply want to make sure that game actions are recorded in chronological order.)

R7. At most one special action move of each of the four types is allowed per game.

R8. If you are recording every action in the game (which I recommend), then the point amounts must be consistent. That is, at the end of move N of a game, the total number of points must be equal to the sum of the total number of points at the end of move $N-1$ (since you must have generated it before) and the number of points awarded for move N.

To make it more straightforward for you to determine how to properly utilize the random number generator for this assignment, we adopt the following conventions:

1. As stated above, there are 10,000 unique user accounts. You are not required to generate games for each of the 10,000 user accounts, this is simply to allow you to create user Ids.

2. A game can be completed in as few as 9 steps, but it can also take as many as 100 steps. An average game lasts around 40-50 steps. Games above 70 steps are **very rare**. You can put together a skewed normal distribution, or use this information in other ways.

3. Moves in the center of the board are slightly more frequent than moves around the edges of the board. Locations (1,20), (20,1), (1,1) and (20,20) should be less frequent than locations (10,10), (11,11),(11,10) and (10,11), for example. I do not provide any specific probability distributions here - figure out and document your own.

4. Special moves are used sparingly. A single game of about 40 moves averages about 2 special moves. Among the special moves, `"Shuffle"` is the most popular, followed by `"Invert"`, `"Clear"` and `"Rotate"`. However, each special move occurs no more than 33% of the time (a special move occurs) and no less than 15% of the time.

5. Multiple people play `BeFuddled` at the same time. Your JSON document shall contain *inteleaved* moves from multiple games. You can determine how many concurrent games are being played at the same time, and how the moves are interleaved, but there should be some

randomness to how this is done (i.e., JSON documents that go Game1-Move1, Game2-Move1,. . . Game100-Move1,Game1-Move2, Game2-Move2,. . . will be considered decifient during the grading process).

6. Assume that you need to start every game for which you want to generate log records. However, you do not need to finish every single game you started. This makes it easy for you to generate exactly the number of JSON objects asked for.

7. All numbering conventions must hold for a single JSON document you are creating, but each run of your JSON generator is considered an independent run, so no need remember the end state of each run. You can start anew each time.

Data Generation for the Thghtshre Messenger

The following rules must be obeyed:

TR1. In a single run, message Ids for individual messages must in increasing order. There is no need to go in a row, but a message Id can be generated only if no other message with a higher id has been generated.

TR2. As mentioned above, if a message object has a `"in-response"` component, the message Id of the `"in-response"` component must be smaller than the message If of the current message object. At the same time, *it is not necessary for you to have generated an actual message with the give `"in response"` message id.*

TR3. Messages with the status `"private"` can have either `"self"` or a user Id as a `recepient`.

TR4. Messages with the status `"protected"` can have either `"self"`, or `"subscribers"`, or a user Id as a recepient.

TR5. Messages with the status `"public"` can have any syntactically legal `"recepient"` value.

In addition to these rules, we adopt the following conventions and provide the following suggestions.

1. There are 10,000 users of the messaging platform. You are not required to generate messages for all of them.

2. Most messages are `"public"`.

3. Most `"protected"` messages are addressed to `"subscribers"`.

4. Most `"private"` messages are addressed to an individual user Id (usually not equal to the user Id of the author of the message).

5. `"Public"` messages are roughly evently distributed between `"all"` and `"subscribers"`, with other options (`"self"`, or a user Id) taking no more than about 20% of the total number of `"public"` messages.

Text Generation for the Messages

Basically, any text generation procedure that uses legible words will suffice.

Here I describe a default procedure you can use. For this lab, I am making available a list of unique English terms used in Jane Austen's *"Sense and Sensibility"* novel. Please note, that this list may include some punctuation, proper names, and ALLCAPS words.

You are not required to generate any meaningful text, beyond generating text that is made out of English words. A simple generation procedure is to generate randomly (using either a normal or a uniform distribution) a number of words a specific message shall contain, and then select the generated number of words at random from the provided list. I am not looking for anything that is significantly fancier than this.

You can use your own word lists if you want to (submit them with your program). Please, no swearing/cursing in this case.

Your messages can be of size 2 to 20 words.

Program Design and Submission

You will create two programs, `beFuddledGen.java` and `thghtShreGen.java` (these, obviously, do not have to be the only files you submit, rather, these are the Java programs that should be compiled and run).

For simplicity, organize your submission as follows. Create two directories, `beFuddled` and `thghtShre` and place all code for each of the generators in the respective directories. You can either submit a single archived directory at a time (but it must be archived from its parent directory, so that a `beFuddled/` or `thghtShre/` directory is proprely created when unpacked), or a single archive that, when unpacked creates two directories.

In addition, you must submit a `README` file (either directly, or as part of an archive that unpacks to the current directory) that describes the following for each of the two programs:

- Any `.jar` files needed to run the program. If you used any `.jar` files that are not part of Java Standard Libraries, or that were not provided on the course web site, it is best to include them in your submission.
- Any compilation instructions and/or run instructions.
- Description of any command-line parameters (see below).
- Description of any specific generation assumptions made. For example, if you decided to use a normal distribution with the mean of 11 and standard deviation of 5 to generate the number of words in a text message, specify this.

I advise you to start building the README file early and to record all necessary information in it as you go.

BeFuddled game. For the `beFuddledGen.java`, the two expected input parameters are name of the output file and the number of JSON objects to generate. You are welcome to add any other input parameters (as optional - if not included, use some documented default values/assumptions). Document all input parameters in the README file. If you need any text/config files to support your program, submit them, and document them in the README file. When the program is run, if it is run without necessary parameters, please provide running instructions/help message.

ThghtShre messenger. For the `thghtShre.java`, the two expected input parameters are the name of hte output file and the number of JSON objects to generate. A possible optional parameter is the word file for message generation (I welcome using it to give your program the flexibility of building messages from a variety of vocabularies). If you are using any word files that are alternative to the one(s) provided by the instructor, submit them. If your word file format is different than the one provided by the instructor, (a) you must submit at least one properly formatted file, and (b) describe the format in the README file[4].

Use `handin` to submit as follows:

Section 01:

```
$ handin dekhtyar lab01-1-01 <FILES>
```

Section 03:

```
$ handin dekhtyar lab01-1-03 <FILES>
```

Good Luck!

[4] As an example, you may want to add frequency of each word/probability of word selection to the file, to be more fancy.